

If-Conversion Optimization using Neuro Evolution of Augmenting Topologies

Reem Elkhoully

Computer Science and Eng. Dept.,
E-JUST, Alex., Egypt
reem.elkhoully@ejust.edu.eg

Keiji Kimura

Dept. of Computer Science and Eng.
Waseda University, Tokyo, Japan
kimura@apal.cs.waseda.ac.jp

Ahmed El-Mahdy

Computer Science and Eng. Dept.,
E-JUST, Alex., Egypt
ahmed.elmahdy@ejust.edu.eg

Abstract

Control-flow dependence is an intrinsic limiting factor for program acceleration. With the availability of instruction-level parallel architectures, if-conversion optimization has, therefore, become pivotal for extracting parallelism from serial programs. While many if-conversion optimization heuristics have been proposed in the literature, most of them consider rigid criteria regardless of the underlying hardware and input programs. In this paper, we propose a novel if-conversion scheme that preforms an efficient if-conversion transformation using a machine learning technique (*NEAT*). This method enables if-conversion customization overall branches within a program unlike the literature that considered individual branches. Our technique also provides flexibility required when compiling for heterogeneous systems. The efficacy of our approach is shown by experiments and reported results which illustrate that the programs can be accelerated on the same architecture and without modifying the original code. Our technique applies for general purpose programming languages (e.g. C/C++) and is transparent for the programmer. We implemented our technique in LLVM 3.6.1 compilation infrastructure and experimented on the kernels of SPEC-CPU2006 v1.1 benchmarks suite running on a multicore system of Intel(R) Xeon(R) 3.50GHz processors. Our findings show a performance gain up to 8.6% over the standard optimized code (LLVM -O2 with if-conversion included), indicating the need for If-conversion compilation optimization that can adapt to the unique characteristics of every individual branch.

Keywords If-Conversion, ILP, Performance Enhancement, Compiler Optimization, Machine Learning, NEAT, Neural Networks.

1. Introduction

Fast performing programs represent a chief goal in the fields of computer architecture and compilers. A key performance driver is providing parallel execution at various levels of granularity. With the multicore shift, larger parallelism granularity is generally sought. In addition to that, single-core performance is still important, as it is a major scaling limiting factor. Accelerating single-core performance mainly relies on instruction-level parallelism (ILP) (Wall 1991), where various independent instructions are executed in the same processing cycle simultaneously on the same CPU (i.e. on different ALUs). The degree of parallelism is inherently limited by the data-flow characteristics of the running program. However, control-flow significantly hinders exploiting the ‘true dependence’ manifested by the data-flow, significantly reducing the achievable degree of parallelism (Allen et al. 1983).

One major approach to alleviating control-flow dependence is branch prediction, where the target of the branch is predicted using one of the multiple conventional branch predictors. Specifically, a branch predictor guesses whether the conditional branch causes

a transfer of control or not (Smith 1981). It relies on the branching outcome history of the branch and/or other spatially contiguous branches. The more complicated the predictor is, the better guessing it can make. That alleviates the speculative execution in modern pipelined processors (Division et al. 1998).

However, not all branches can be easily predicted, particularly those with a random outcome (such as branches relying on element comparisons when sorting random inputs). Mispredictions come with high time penalty resulting from the wasted processing cycles in executing the wrongly predicted instructions, flushing them and fetching the correct ones (Hennessy and Patterson 2011). Another argument that makes the branches undesirable is that they limit the instruction fetch bandwidth (Shen and Lipasti 2013) which diminishes the instruction level parallelism (ILP), hence the number of instructions executed per cycle.

The if-conversion approach is used to convert control-dependence into data-dependence, aiming to defeat the above performance obstacles. In this model, instructions are guarded by ‘predicates’, if supported by the hardware otherwise conditional moves are used, thereby eliminating control-flow (Mahlke et al. 1995). This approach thus relies on ‘if-conversion’ optimizations to convert conditional branches into predicated instructions, allowing for further potential parallelization subject to the inherent data-flow dependences. However, predication comes at the extra cost of executing ‘nullified’ instructions. That can potentially degrade performance for large ‘if-then’ bodies. Moreover, branches interact in terms of allowing for different execution schedules, for which finding the optimal schedule is a hard combinatorial search problem where its complexity grows exponentially with the number of branches per function in the program.

In this paper, we revisit the problem of deciding which branches to convert. In particular, we implement the machine learning method *NEAT* (Stanley and Miikkulainen 2002) in the LLVM compiler to replace the heuristics used in if-conversion optimization. The advantage of using this algorithm is the ability to evaluate the if-conversion for all branches in the program taking into consideration mutual influence. *NEAT* fits for this problem because of its high performance in searching large spaces (exponential in number of branches in this problem). Our system successfully achieved up to 8.6% performance gain transparently on the same architecture and does not require altering the original code by any mean. Our technique identifies code features that characterize different branches in the program. These features are fed to the machine learning algorithm *NEAT* during the if-conversion optimization to customize the transformation plan.

Our contributions can be summarized in three basic modules:

- First, a module that analyzes the code to capture code features as a vector for each branch.

- Second, a module that applies the *NEAT* algorithm to repeatedly customize the if-conversion decision for all the branches as a vector of ones and zeros which we call the **bitmask** and eventually provides the best performing optimized program.
- Third, a module that controls the if-conversion optimization in the LLVM according to the bitmask generated by the second module.

We consider C/C++ kernels from the SPEC-CPU2006 v1.1 benchmarks suite (Standard Performance Evaluation Corporation) for the experiments. The results are comprehended in comparison to the LLVM’s compilation behavior on the same kernels.

The rest of this paper is organized as follows; Section 2 explains essential background. Section 3 provides detailed description for our system and contributions. Section 4 presents and discusses results. Section 5 provides related work. Finally, Section 6 concludes the paper and discusses future work.

2. Background

In this section, we provide the related background that is necessary to illustrate our system. Section 2.1 presents various if-conversion transformations. Section 2.2 introduces the LLVM if-conversion technique and how we use it to build our analysis module. Section 2.3 explains the machine learning algorithm we used for our system (*NEAT*).

2.1 If-Conversion Optimizations

Control-flow optimizations, such as branch optimizations, relax the control dependences in the program in order to facilitate instruction parallelization and speculative execution. Conditional branches add control-flow dependences to the program, as their outcome is not known until runtime. Systematically converting control dependence to data dependence was initially introduced by Allen et al. (Allen et al. 1983). The if-conversion process aims to remove branches from a given program (Allen and Kennedy 2002). Certainly, a branch cannot be deleted without being replaced by another control that maintains program proper functionality. That could be accomplished either by conditional moves (CMOVs) or guarded (predicated) execution.

Conditional Moves (CMOVs): They are hardware supported instructions that copy a value from a source to a destination if and only if a specific condition is true. This condition is assessed, and then one of the special purpose registers (i.e. EFLAGS register in x86) is changed. That enables the compiler to convert a piece of code that contains simple branches into branch free code. Several processor architectures, such as the SPARC architecture starting from SPARC-V9 and Pentium Pro (Muchnick 1997) and more recent processors, support conditional moves.

Guarded (Predicated) Execution: It removes forward branches by adding guard expressions that control the execution of the instructions. The guard is a Boolean expression that is constructed by joining a set of conditions; each of these represents a conditional branch that leads to the current instruction. It is obvious that the guard expression may grow complex if the instruction is control-dependent on many conditional branches. The guard expression could be simplified, but the problem of simplifying a Boolean expression is NP-Complete. Some processor architectures such as Intel IA-64 and ARM 32-bit support predicated execution by a fully predicated instruction set using special registers. To ensure the correctness of the program after branch removal, three constraints should apply (Allen and Kennedy 2002): (1) The guard expression of an instruction is true if and only if the corresponding statement in the original program is executed. (2) The order of execution of instructions that have true guards in the new program is the same as their order of execution in the original program. (3) Any expres-

sion with side effects is evaluated exactly as many times in the new program as the original program. If-conversion optimizations are of surpassing importance because conditional branch instructions are very expensive –in terms of runtime– when mispredicted, besides, being obstacles in the way of *ILP*.

We propose a technique for tuning if-conversion optimization - using LLVM - valid for any architecture that supports predicated execution. our technique uses a machine learning algorithm that is trained to tune if-conversion to give the best performance for the whole compiled program. When compiles for heterogeneous systems, different compilations are done for different underlying hardware ensuring a suitable optimization tuning for each of them. While the effect of branches and branch predictors differ among the architectures, our technique which is adaptable will be of great value. The following section studies the utility of these optimizations in the LLVM if-conversion module, then, how we used it in our technique.

2.2 LLVM If-conversion

The LLVM (The LLVM Compiler Infrastructure) is a compiler infrastructure that conducts code analysis and transformation using reusable modules, called passes, that support static and dynamic compilation. It includes a wide range of capabilities that appear transparent to programs. Moreover, it provides high-level information at compile-time, link-time, runtime and between runs. Besides, LLVM provides a large number of analysis and optimization tools that enable precisely generating the object code in a way that serves specific endeavors. In this infrastructure, a low-level Static Single Assignment (SSA) form is elaborated for code representation (Lattner and Adve 2004). LLVM provides if-conversion optimization either for the architectures that support predicated instructions or the out-of-order CPUs, where both the “then” and “else” bodies are executed, and the result is chosen by a *cmov* instruction. Therefore, branches that may be mis-predicted are omitted.

The LLVM if-conversion transformation is included in the -O2 optimization level and can also be separately requested using command line options. For every function, it traverses the whole dominator tree of basic blocks post-orderly. All instructions within the basic block should be valid for speculative execution in order to allow conversion. For the nested branches, the conversion starts from the most inner tracing back up to the outer ones. To avoid critical edges, the if-conversion pass handles either triangle or diamond branches only. Another conversion constraint is that the considered block should end with a convertible to “select” instruction, *PHI* node. All of these measures would assure the program’s correctness.

After checking the eligibility of if-conversion for a basic block, a heuristics based cost model is applied to decide if the conversion is profitable. The profitability is measured in terms of multiple criteria: First, the critical path should not be extended by more than a certain limit, chosen to be half of the misprediction penalty. Second, if-converted trace length shortening that is limited by ILP resources should exceed that of the critical path in addition to the misprediction before conversion. Finally, the delay to “select” instruction by data dependence in both “then” and “else” traces should be considered. In our system, we replace the cost model and heuristics by a machine learning algorithm (*NEAT*) that decides the if-conversion profitability according to the code features and a corresponding fitness value reflects the performance of the whole program.

2.3 NEAT

Neural networks are famous for their ability to model nonlinear problems of high complexity. NeuroEvolution of Augmenting Topologies (*NEAT*) is a machine learning algorithm that artificially evolves neural networks (NNs) using genetic algorithms tech-

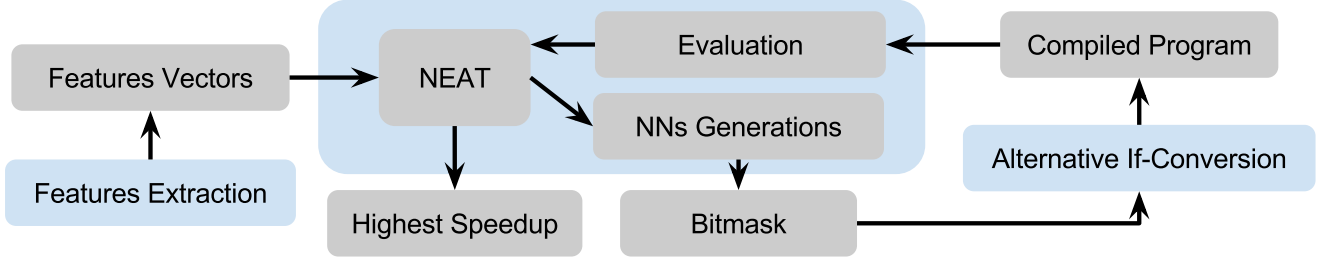


Figure 1: System architecture that mainly consists of three modules.

niques. It does not only evolve the weights of connections, but also evolves the structure of the network. *NEAT* is proved to outperform fixed-topology techniques and showed promising performance in the reinforcement learning problems (Stanley and Miikkulainen 2002).

NEAT starts from an initial generation of minimal uniform NNs where the input nodes are connected directly to the output nodes with zero hidden nodes. Genetic encoding is used where genomes linearly represent NNs. Thus a genome is a list of connection genes, each connecting two node genes and specifies the corresponding weight. *NEAT* allows a mutation that can change connection weights and network structure (adding/deleting nodes and/or connections) to generate a population of NNs. The outputs of these networks are evaluated through a fitness function, then the best of them are selected to be the parents for the next generation of networks (Stanley and Miikkulainen 1996). This process is repeated over successive generations of NNs, preserving dimensionality minimization through minimal structure incremental growth, until the desired fitness is found. In our system, we use *NEAT* as a machine intelligence technique to search the space of 2^n possible combinations of converted and non-converted n branches in the program in order to find the combination that achieves the highest performance.

3. System Description

Our system customizes the if-conversion optimization replacing the fixed heuristics used in literature to estimate the profitability of the if-conversion by a machine learning strategy based on *NEAT*. It works for any program written in any language which is supported by the LLVM compiler and does not require any changes or additions to the original code of the program. We apply *NEAT* to a set of descriptive features that precisely characterizes the code of *Basic Blocks (BBs)* that contain different branches in the program. For this purpose, our system is implemented in a three major modules illustrated in Figure 1 which are implemented as plugins in the compiler infrastructure.

Module 1 is responsible for inspecting the code to find the branches that can be converted safely without affecting the correctness of the program, then extracts the set of features listed in Table 1 for each one of them. These features are stored in vectors then forwarded to the next module.

Module 2 receives the features vectors and pushes them as an input for the *NEAT* initial population of NNs. Each NN is responsible for evaluating the features of a single branch to decide whether it shall be if-converted. This decision is expressed as a single NN output (1/0). The output of all NNs - a stream of 1's/0's which we call a **bitmask** - is sent to the third module that controls the if-conversion of the corresponding branches on/off. The program is compiled with the customized if-conversion optimization, executed and evaluated against the fitness function which we designed as

the speedup over the program when optimized with the original if-conversion of the LLVM. A group of the NNs that gave the best performance are chosen to be the parents of the next generation of NNs. A new generation of NNs is generated from those parents using GA which enhances the structure of NNs. *NEAT* continues to iterate over new generations to maximize the fitness (speedup) until it reaches a defined limit of number of generations.

Module 3 is an alternative for the original if-conversion pass in LLVM. This module receives the **bitmask** for every program compilation where the number of bits equals the number of branches inspected. The if-conversion for each branch is turned on/off according to the value of the corresponding bit. This if-conversion is done within the whole compilation process, the output program is run and the speedup is calculated. The overall system eventually reports the highest speedup, the corresponding NN, the bitmask and the best performing compiled version of the program.

Our system is convenient for compiling for heterogeneous systems requires consideration to its special nature. Exploiting available resources as the hardware configuration changes is essential to make full utilization of them. The compiler should be aware of generating code that targets different hardware. Our technique adjusts if-conversion optimization to suit targets. As it does not consider rigid hardware specifications as a metric for the optimization tuning. On the contrary, the tuning is done through learning from several runnings. That allows the tuner to adapt with whatever existing underlying hardware.

4. Experimentation

We constructed a testbed for our if-conversion using *NEAT* to customize the optimization. The operating system is a server version of Ubuntu 14.04 LTS running on top of a multicore system of Intel(R) Xeon CPU E5-2637 v3, which has a speed of 3.5 GHz per core. As for the memory modules, it has a capacity of 64 GB and access rate of 2133 MHz.

For accurate measurement of time we used Intel's Time Stamp Counter (Int) to count processing cycles consumed by the test programs. The implementation of our study methodology was based on the LLVM compilation framework version no. 3.6.1 with our extension that is applicable to any programming language supported by LLVM, and requires no change/addition to the original code of the program.

We set *NEAT* to form 30 NNs per generation and to iterate for 50 generations. These parameters were tuned as a trade-off between the training time -which is proportional to the size of the experiment- and the performance improvement achieved. As for the test subject, we considered the integer benchmarks from SPEC-CPU2006 v1.1 (Standard Performance Evaluation Corporation) executed with *reference* workload.

As shown in Figure 2, our system improves the performance with a percentage ranging from 0.74% to 8.6% for all inspec-

Feature	Description
Basic Block (BB) size	No. of instructions in the <i>Basic Block (BB)</i> that contains the branch.
True BB Critical Path	Critical Path length -data dependency- to the True <i>BB</i> .
False BB Critical Path	Critical Path length -data dependency- to the False <i>BB</i> .
Minimal Critical Path	Minimum of the <i>True</i> and <i>False</i> Critical Paths.
Unexploited ILP	Maximum <i>ILP</i> achieved within the <i>BB</i> after if-conversion.
Branch Depth	The distance of the earliest issue cycle as determined by data dependences and latencies from the beginning of the trace.
Loop Depth	The nesting level of the loop that contains the <i>BB</i> .
Slack Sum	No. of <i>Cycles</i> the instruction can be delayed without increasing <i>Critical Path</i> .
Maximum depth	The <i>slack</i> of the <i>PHI</i> node in addition to its depth from the tail trace beginning.
True BB depth	The depth of the <i>PHI</i> node from the True <i>BB</i> + <i>PHI True Cycles</i> .
False BB depth	The depth of the <i>PHI</i> node from the False <i>BB</i> + <i>PHI False Cycles</i> .

Table 1: Code features used by *NEAT* to customize if-conversion

ted benchmarks over the original if-conversion optimization of LLVM on the same architecture. In Table 2 we listed the number of branches inspected that can be if-converted while preserving the correctness of the program in every benchmark, in addition to the speedup achieved. As expected, the larger the number of candidate branches, the higher the speedup achieved. *403.gcc* is on the top of list with 4804 branches and 1.086 speedup; on the contrary (*429.mcf*) comes at the end with 12 branches and 1.007 speedup. Moreover, this proportional behavior applies for all the benchmarks except *445.gobmk* and *458.sjeng* which give lower speedups than expected. This is an indication that most of the candidate branches are not part of the frequently executed code which keeps the speedup lower than expected. On the other hand *462.libquantum* gives a much higher speedup compared to the number of candidate branches it contains. This is a clue that these candidate branches are very effective and they have a major influence on the frequently executed code.

5. Related Work

The if-conversion related work in the literature is distinguishable by whether if-conversion is performed statically, dynamically, or hybrid. While the previous compiler machine learning-aided techniques target various problems such as the optimal selection of optimizing transformations, their ordering and the best values for the transformation parameters.

Static if-conversion depends principally on information such as the misprediction penalty and the number of cycles within the *if* body; that can be collected in an offline analysis (profiling) before runtime. There are many techniques in the literature that adopt static if-conversion which are described below. A compilation framework that delays the if-conversion to schedule time is designed to allow the compiler to minimize runtime by balancing the control-flow and predicated branches (August et al. 1997). The authors in (Mantripragada and Nicolau 2000) present an algorithm to perform if-conversion selectively on out-of-order processors that support dynamic speculation and guarded execution. They identified three criteria to measure the profitability of the conversion namely based on size, predictability and profile. The effect of their technique on the net cycles, mispredictions and mis-speculation is exhibited in their paper.

Another algorithm is reported in (Noah Snaveley 2002) for the Itanium architecture; it initially operates on unpredicated code, and the if-conversion optimization is performed late in the compilation process. That generates faster (less runtime) and denser (fewer instructions) code.

Further, an algorithm that minimizes the number of predicates assigned to basic blocks, which are assigned as early as possible using dominance relations to relax dependence constraints, is shown in (Fang 1997). Moreover, in (August et al. 1999) the program control-flow is represented as a graph (called program decision logic network), then it is modeled in a Boolean equation, which is then minimized and used to regenerate predicated code.

An algorithm that uses dynamic programming to generate code for different target architectures that support predicated execution is discussed in (Hohenauer et al. 2008). Another approach handcrafts well-known algorithms (i.e. sorting and searching) into a constant number of branch-free loops, such that a branch predictor can achieve $O(1)$ mispredictions (Elmasry and Katajainen 2012).

Other methods use dynamic if-conversion in which a profiling process during runtime is used to capture some characteristics (e.g. misprediction rate) to be used in optimization. In (Hazelwood and Conte 2000), runtime information is used to construct a dynamic optimizer that complements the static one in a previously presented algorithm. It can convert branches, or reverse their conversion, targeting the higher performance based on profiling the program to discover the highly mispredicted branches. Although this algorithm chooses the conversions that improve performance, it does not consider any correlation between the different *ifs* which are most probably related to each other especially within the same function.

Also, the authors in (Jordan Alexander 2013) present two algorithms for if-conversion: one of them targets the intermediate representation (IR) level and the other targets the machine code level. Two heuristics are used to calculate the profitability of the conversion. The execution time (number of cycles in the basic block instructions) based on the sizes of basic blocks and basic blocks execution frequencies.

Also, a hardware that uses runtime information to choose to convert only the hard-to-predict branches is presented in (Kim et al. 2006). The presented system provides two versions of the code: one can be predicate executed and the other using a branch predictor.

Finally, there are methods that combine both static and dynamic methods: A tree-based model to make predictions using predication and vectorization techniques are presented in (Asadi et al. 2012); it introduces runtime performance ranking assuming that a trained model already exists. Meanwhile, data is laid out in memory in an architecture-conscious method. Random features IDs, thresholds, and regression values are used to generate features values in a features vector.

Another contribution is proposed in (Jimenez and Lin 2001), where a simple neural network (perceptron) hardware implementation is provided to improve branch prediction. Another method (Calder et al. 1997) uses program-based static branch prediction

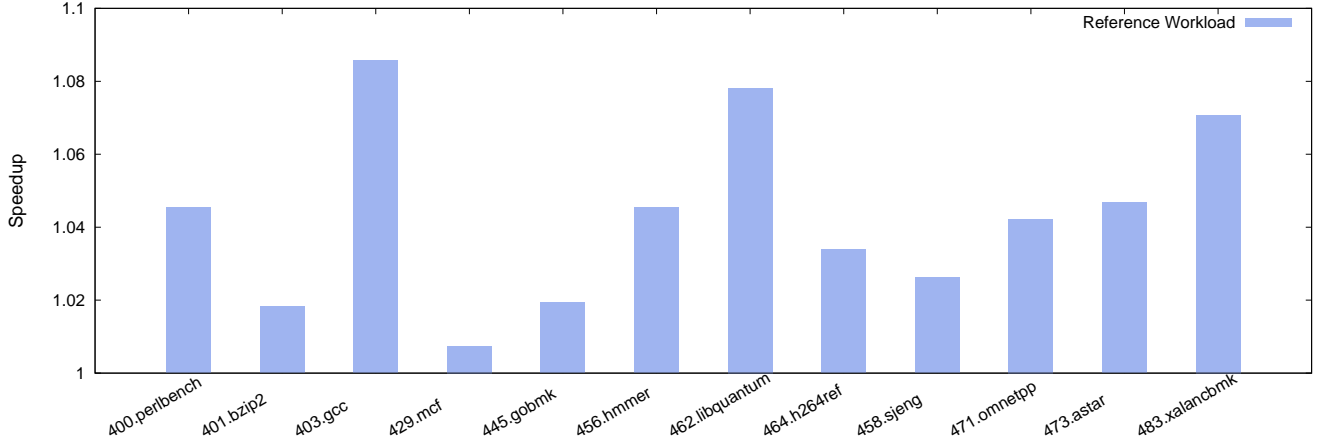


Figure 2: Speedup of benchmarks

Benchmark	Description (Henning 2006)	No. Branches	Speedup
400.perlbench	A cut-down version of Perl v5.8.7	717	1.045
401.bzip2	Compression software based on Julian Seward's bzip2 version 1.0.3	41	1.018
403.gcc	C Language optimizing compiler	4804	1.086
429.mcf	Combinatorial optimization - Singledepot vehicle scheduling	12	1.007
445.gobmk	Artificial Intelligence - Go game playing	1000	1.019
456.hmmer	Search a gene sequence database - Profile HMMs	166	1.045
458.sjeng	Artificial Intelligence (game tree search - pattern recognition)	172	1.026
462.libquantum	Physics - Quantum Computing Simulation	25	1.078
464.h264ref	Video compression	652	1.034
471.omnetpp	Discrete Event Simulation of a large ethernet network	102	1.042
473.astar	Computer games - Artificial Intelligence - Path finding	57	1.047
483.xalancbmk	Processor for transforming XML documents into HTML, text, or other XML document types	934	1.071

Table 2: Candidate Branches and Speedup.

based on neural networks and decision trees to map static features associated with each branch to a prediction.

Moreover, an attempt to construct an online ensemble learning framework consisting of small trees to solve the problem of hardware conditional branch prediction is discussed in (Fern and Givan 2003). The learning based techniques (the third category) may be the only one that considered the effect of if-conversion on the whole program, as they train their systems over iterations to converge to the least possible runtime. Others make a separate decision for individual *if*'s regardless of the correlation with other ones. But, the problem is how fast they can evolve the selection space towards the best performing and that is what we handle in our system.

As an example for research that considered using machine learning algorithms for efficient compiler optimization, ordering or transformation parameters tuning, we demonstrate some of them below. A compiler based approach for mapping parallelism to multicore processors is proposed in (Wang and O'Boyle 2009). This technique applies an off-line trained *NNs* to develop a number of threads and scheduling predictors for parallel programs. An attempt of speeding up iterative compilation -through building models of the program features using machine learning techniques- was presented in (Agakov et al. 2006). Authors in (Monsifrot et al. 2002) introduced a decision-tree based technique that generates compiler heuristics for the target processor considering the loop unrolling optimization as a source of performance features. (Yotov et al. 2003). A per-method logistic regression technique is

used to select proper optimizations for each method in the program depending on its features in (Cavazos and O'boyle 2006).

6. Conclusion

If-conversion is a compiler optimization that tackles code performance. It seeks to increase the instruction fetch bandwidth and to decrease branch mispredictions. That can be done by substituting the branches by branch free pieces of code.

This paper investigates the efficacy of a novel technique that uses a machine learning approach that evolves *NNs* to solve problems with high complexity, namely *NEAT*; the technique customizes if-conversion in one of the recently developed and rapidly grown open source compilers (LLVM). We examine the adequacy of our system for The INT suite of kernels from the SPEC-CPU2006 v1.1 benchmarks. The experiments are performed by extending the LLVM compilation framework to provide for such analysis. The results show notable discrepancy between optimized code using LLVM built-in technique and the best configuration achieved by our system, more than 8.6% in *462.libquantum*. Moreover, our system does not require neither changes nor additions to the original code of the program.

Future work would consider dynamic compilation analysis taking into consideration behavior change between calls. On the other hand, we are going to expand our research to include architectures that support predicated execution such as the ARM processors.

7. Acknowledgement

We are grateful to Prof. Hironori Kasahara and Prof. Kazunori Ueda, Department of Computer Science and Engineering, Waseda University, Tokyo, Japan, for their contributions, discussions and advice.

References

- F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.
- J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7. doi: 10.1145/567067.567085. URL <http://doi.acm.org/10.1145/567067.567085>.
- R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.
- N. Asadi, J. Lin, and A. P. de Vries. Runtime optimizations for prediction with tree-based models. *CoRR*, abs/1212.2287, 2012.
- D. August, W.-M. Hwu, and S. Mahlke. A framework for balancing control flow and predication. In *Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pages 92–103, 1997. doi: 10.1109/MICRO.1997.645801.
- D. August, J. Sias, J.-M. Puiatti, S. Mahlke, D. Connors, K. Crozier, and W.-M. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 208–219, 1999. doi: 10.1109/ISCA.1999.765952.
- B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, Jan. 1997. ISSN 0164-0925. doi: 10.1145/239912.239923. URL <http://doi.acm.org/10.1145/239912.239923>.
- J. Cavazos and M. F. O'Boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, 41(10):229–240, 2006.
- I. B. M. C. R. Division, P. Raghavan, H. Schachnai, and M. Yaniv. *Dynamic Schemes for Speculative Execution of Code*. Computer science. IBM Research Division, 1998. URL <https://books.google.co.jp/books?id=eBgMGwAACAJ>.
- A. Elmasry and J. Katajainen. Lean programs, branch mispredictions, and sorting. In *Fun with Algorithms*, volume 7288 of *Lecture Notes in Computer Science*, pages 119–130. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30346-3. doi: 10.1007/978-3-642-30347-0_14.
- J. Fang. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In *Languages and Compilers for Parallel Computing*, volume 1239 of *Lecture Notes in Computer Science*, pages 135–153. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63091-3. doi: 10.1007/BFb0017250. URL <http://dx.doi.org/10.1007/BFb0017250>.
- A. Fern and R. Givan. Online ensemble learning: An empirical study. *Machine Learning*, 2003. ISSN 0885-6125. doi: 10.1023/A:1025619426553.
- K. M. Hazelwood and T. M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Parallel Architectures and Compilation Techniques Proceedings.*, pages 71–80. IEEE, 2000.
- J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
- M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, H. Meyr, G. Bette, and B. Singh. Retargetable code optimization for predicated execution. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1492–1497. ACM, 2008.
- Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference. Intel.
- D. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *The Seventh International Symposium on High-Performance Computer Architecture, HPCA.*, pages 197–206, 2001. doi: 10.1109/HPCA.2001.903263.
- K. A. Jordan Alexander, Kim Nikolai. Ir-level versus machine-level if-conversion for predicated architectures. In *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems, ODES '13*, pages 3–10. ACM, 2013. ISBN 978-1-4503-1905-8. doi: 10.1145/2443608.2443611. URL <http://doi.acm.org/10.1145/2443608.2443611>.
- H. Kim, O. Mutlu, Y. N. Patt, and J. Stark. Wish branches: Enabling adaptive and aggressive predicated execution. *Micro, IEEE*, 26(1):48–58, 2006.
- C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, CGO.*, pages 75–86. IEEE, 2004.
- S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149. IEEE, 1995.
- S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *Proceedings of the 14th International Conference on Supercomputing, ICS*, pages 206–214. ACM, 2000. ISBN 1-58113-270-0. doi: 10.1145/335231.335251. URL <http://doi.acm.org/10.1145/335231.335251>.
- A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer, 2002.
- S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.
- G. A. Noah Snaveley, Saumya Debray. Predicate analysis and if-conversion in an itanium link-time optimizer. In *Proceedings of the Workshop on Explicitly Parallel Instruction Set Architectures and Compilation Techniques (EPIC-2)*, 2002.
- J. P. Shen and M. H. Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.
- J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.
- Standard Performance Evaluation Corporation. SPEC Benchmarks, <http://www.spec.org/>. URL <http://www.spec.org/>. Accessed Sept.1,2015.
- K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. *Network (Phenotype)*, 1(2):3, 1996.
- K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL <http://nn.cs.utexas.edu/?stanley:ec02>.
- The LLVM Compiler Infrastructure. <http://www.llvm.org/>. URL <http://www.llvm.org/>. Accessed Sept.1,2015.
- D. W. Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.
- Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Sigplan notices*, volume 44, pages 75–84. ACM, 2009.
- K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Notices*, volume 38, pages 63–76. ACM, 2003.